Computer Science Department

TECHNICAL REPORT

Worst Case Data Structures for the Priority Queue with Attrition

R. Sundar

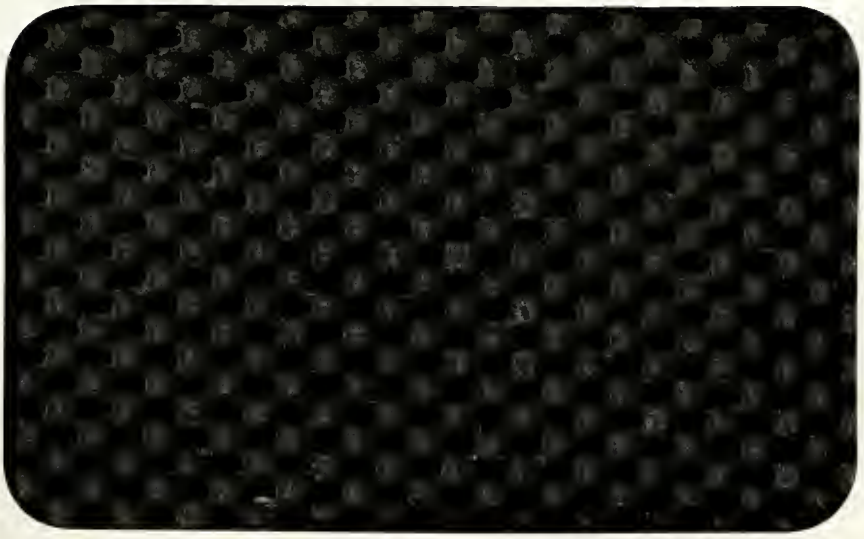
Technical Report 385

June 1988

NYU COMPSCI TR-385 c.2
Sundar, Rajamani
Worst case data structures for the priority queue with attrition.

NEW YORK UNIVERSITY

Department of Computer Science
Courant Institute of Mathematical Sciences
251 MERCER STREET, NEW YORK, N.Y. 10012



Worst Case Data Structures for the Priority Queue with Attrition

R. Sundar

Technical Report 385

June 1988

This work was supported in part by NSF grant CCR-8702271 and by ONR grant N00014-85-K-0046.

Worst Case Data Structures for the Priority Queue with Attrition

Rajamani Sundar[0]
Courant Institute of Mathematical Sciences
New York University

May 30, 1988

Abstract

We describe three data structures for the Priority queue with attrition (or PQA), that perform each PQA operation in $O(1)$ worst case time. Previous implementations of the PQA required $O(1)$ amortized time per operation.

1 Introduction

A *Priority queue with attrition* (*PQA* for short) is a set of items, each having a real valued key, supporting the following operations :

CREATEPQA: Create an empty PQA.

DELETEMIN: Delete and return the minimum element of the PQA.

INSERT(x): Insert a new item x into the PQA, deleting all items larger than or equal to x from the PQA.

Additionally, we might want to perform the following operations on the PQA:

FINDMIN: Return the minimum item in the PQA.

DELETE(x): Given the position of item x in the data structure, delete x.

FINDMIN can be easily incorporated into any data structure for the PQA supporting the first three operations. This is done by maintaining the minimum element separately and representing the remaining elements of the PQA using

[0]This work was supported in part by NSF grant CCR-8702271 and by ONR grant N00014-85-K-0046.

the data structure. Guaranteeing an $O(1)$ time bound for DELETE is usually more difficult.

The priority queue with attrition crops up in a variety of applications. The simplest instance of its use is in implementing a *minqueue*, which is nothing but a queue with a FINDMIN operation, besides insertions at the rear and deletions at the front. To perform FINDMIN efficiently, we maintain a sequence $x_1, x_2, \ldots, x_k$ of items, such that

1. x_1 is the smallest item in the queue, and
2. $\forall_{1 \leq i < k}\ x_{i+1}$ is the smallest item to the rear of x_i.

When a new item x is inserted, if $x_i < x \leq x_{i+1}$, then the items $x_{i+1}, x_{i+2}, \ldots, x_k$ are all discarded, and x is appended to the sequence. Thus, the sequence behaves like a PQA, and our data structures for the PQA provide implementations of the minqueue requiring $O(1)$ time per operation. Gajewska and Tarjan [3] offer a different implementation having the same time bound for the minqueue operations. PQAs also find application in pagination and river routing [1,2,4,5].

It is easy to implement the PQA using $O(1)$ amortized time per operation [1,4]. Maintain the items of the PQA in a queue ordered according to arrival time. This causes the queue to be also ordered by key. Hence, DELETEMIN simply removes and returns the first element of the queue. INSERT(x) scans the queue from the rear, deleting all items greater than or equal to x, and inserts x at the rear. Since an item can get deleted at most once, the data structure spends only $O(1)$ amortized time per operation.

Is there a data structure for the PQA that requires only $O(1)$ time per operation in the worst case? We answer the question in the affirmative by providing three $O(1)$ worst case time data strutures for the PQA. Section 2 describes a data structure with $O(1)$ amortized time per operation, which is an useful stepping stone in the derivation of the worst case data structures. In sections 3, 4, and 5, we derive the worst case data structures by modifying the amortized data structure. The last section discusses a related problem. We introduce some definitions and notations in the rest of this section.

An item of a queue is *clean*, if it is smaller than all the items to its rear, and *dirty*, otherwise. Call a queue *clean*, if all its items are clean, *i*.e. the items increase in key from the front to the rear. We say a queue Q *represents* a set of items S, if S equals the set of clean items of Q. For any two queues Q_1 and Q_2, by $Q_1 < Q_2$, we mean that the items in Q_1 are smaller than the items in Q_2.

For any queue $Q = (q_1, q_2, \ldots, q_n)$ with q_1 as the first element, let

$$\text{first}(Q) = q_1$$

$$\text{last}(Q) = q_n$$

$$\text{rest}(Q) = (q_2, q_3, \ldots, q_n)$$

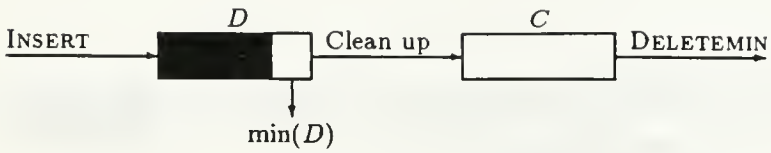


Figure 1: The amortized data structure. Darkening signifies the lack of key-order, and the arrows denote the data paths for the movement of items.

$\min(Q)$ = a smallest item of Q

$Q \parallel (q_{n+1}, \ldots, q_{n+m}) = (q_1, \ldots, q_n, q_{n+1}, \ldots, q_{n+m})$

$|Q| = n$.

Define the queue operations

DELETEFIRST(Q): Delete and return first(Q).

DELETELAST(Q): Delete and return last(Q).

PASS(Q_1,Q_2): Pass first(Q_1) to the rear of Q_2.

PASSBACK(Q_1,Q_2): Pass back last(Q_1) to the front of Q_2.

Each queue operation can be performed in $O(1)$ time, if we represent a queue by a doubly linked list with pointers to the first and last nodes.

The concurrent assignment statement

$$v_1, v_2, \ldots, v_n := e_1, e_2, \ldots, e_n$$

first evaluates the expressions $e_1, \ldots, e_n$ and then assigns their values to the variables $v_1, \ldots, v_n$.

2 The amortized data structure

The amortized data structure has a clean queue C, from which the minimum element is always deleted, and a "dirty" queue D, to which new items are added. Whenever DELETEMIN finds the minimum item of the PQA in D, a *clean up* process recreates C from D, by stripping off dirty items. See figure 1.

Formally, the following invariants are maintained:

1. $C \parallel D$ represents the PQA.
2. C is clean.
3. first(D) = min(D).

The operations are implemented as follows:

```
CREATEPQA ≡
    C, D := (), ()

INSERT(x) ≡
    if D ≠ () cand first(D) ≥ x  then
      { Delete all existing items of D; add x to D }
          D := (x)
    else D := D || (x)

DELETEMIN ≡
    { Ensure that C ≠ (), and
                    D ≠ () ⇒ first(C) < first(D) }
      if (D ≠ () and C ≠ ()) cand first(C) ≥ first(D)  then
        { Destroy C }
            C := ()
      if C = ()  then
        Clean up D and transfer its contents to C.
    return DELETEFIRST(C)
```

D can be cleaned up in $O(|D|)$ time in a single forward or backward scan. The forward scheme scans D from the front to the rear, and at each item x, deletes larger or equal items to the front of x. The backward scheme scans D in reverse, deleting each item with a smaller or equal successor. Since clean up costs only $O(1)$ time per item, this data structure requires only $O(1)$ amortized time per operation. DELETE simply marks the deleted item; marked items are deleted when they appear at the front of C.

3 Worst case data structure 1

We convert the amortized data structure into a worst case data structure by overlapping the PQA operations with the clean up operation, and systematically transferring clean items from the dirty queue to the clean queue. We also ensure that the minimum element of the PQA is at the front of the clean queue as far as possible. In this section, we apply the forward scheme for clean ups, and derive our first data structure requiring $O(1)$ time per operation. The worst case data structure presented in the next section is derived by using the backward clean up scheme to clean up the dirty queue.

During the forward clean up scheme, the dirty queue gets divided into a clean front portion D_f, and a rear portion D_r, that is waiting to be cleaned up. At the beginning, $D_f = ()$, and $D_r = D$. Each clean up step either transfers first(D_r) to the rear of D_f, or deletes last(D_f), depending on whether last(D_f) < first(D_r), or not. At any stage, the algorithm needs to perform at

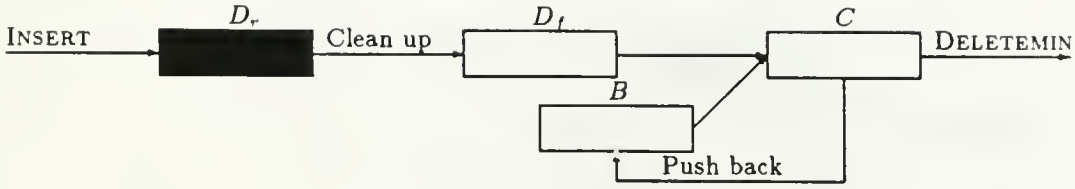


Figure 2: Worst case data structure 1

most $2|D_r| + |D_f|$ clean up steps before it can finish. In order to always have the minimum element of the PQA at the front of C, we impose the constraints that C contain only items smaller than the items of the other queues, and that $|C| \geq 2|D_r| + |D_f|$. New items are usually appended to D_r. Inserting an item that is smaller than some of the items of C will then violate the minimality of the items in C. This problem is resolved by pushing back the items in rest(C) into a buffer B, and retaining first(C) in C, after checking that first(C) is smaller than the inserted item. Whenever $D_r = ()$, items are successively transferred from B to C, until first(B) $\geq$ first(D_f). Then, B is emptied, and D_f is appended to C. See figure 2.

More formally, the worst case data structure maintains the queues C,B,D_f, and D_r under the following invariants:

1. $C \parallel B \parallel D_f \parallel D_r$ represents the PQA.
2. C,B, and D_f are clean, and $C < B \parallel D_f \parallel D_r$.
3. **Bias:** $|C| \geq 2|D_r| + |D_f|$.

INSERT and DELETEMIN can upset the bias, which is restored by performing the following BIAS operation as many times as necessary:

```
BIAS ≡
    if D_r ≠ ()  then
      { Clean up step }
        if D_f ≠ () cand last(D_f) ≥ first(D_r)  then
          DELETELAST(D_f)                          { decrease |D_f| }
        else PASS(D_r,D_f)          { decrease |D_r|; increase |D_f| }
    else if D_f ≠ () cand (B = () cor first(B) ≥ first(D_f))  then
      D_f, B, C := (), (), C ∥ D_f      { decrease |D_f|; increase |C| }
    else if B ≠ ()  then PASS(B,C)                  { increase |C| }
    { else B = D_f = D_r = () }
```

BIAS increases $|C| - 2|D_r| - |D_f|$ by at least 1, if any of the queues B,D_f, and D_r is nonempty. This is accomplished by either performing a clean up step, or passing an item from B to C, or appending the items of D_f to C. Hence, an INSERT must be followed by two BIASs and a DELETEMIN by one BIAS.

The implementation of the PQA operations is now straightforward:

```
CREATEPQA ≡
    C, B, D_f, D_r := (), (), (), ()

INSERT(x) ≡
    if C ≠ () cand first(C) ≥ x then
      { Delete all existing items; add x to C }
        C, B, D_f, D_r := (x), (), (), ()
    else if C ≠ () cand last(C) ≥ x then
      { Empty B,D_f, and D_r; Push back rest(C) into B; add x to D_f }
        C, B, D_f, D_r := (first(C)), rest(C), (x), ()
    else D_r := D_r || (x)
         BIAS; BIAS

DELETEMIN ≡
    BIAS
    return DELETEFIRST(C)
        { BIAS ensures that C ≠ (), unless the PQA is empty }
```

It is evident that the implementation of the operations is correct, and that each operation requires only $O(1)$ time in the worst case. In practice, instead of maintaining the queues C,B,D_f, and D_r as four separate lists, it is more efficient to maintain a single doubly linked list representing $C \parallel B \parallel D_f \parallel D_r$ with pointers to first(C), first(B), first(D_f), first(D_r) and the last node.

How is DELETE handled? If the deleted item belongs to B or C, it is immediately deleted, and one BIAS is performed if it was deleted from C. If it is in D_f or D_r, the item is just marked, and is deleted only when it enters C. This calls for a modification of BIAS: where D_f is appended to C, instead of appending the whole of D_f, it suffices to move just one item to C, namely first(D_f), and if first(D_f) is marked, it is deleted instead.

4 Worst case data structure 2

In the course of the backward clean up scheme, the dirty queue is partitioned into a clean rear portion D_r, and a front portion D_f, awaiting cleaning. To start with, $D_r = ()$, and $D_f = D$. Each clean up step either transfers last(D_f) to the front of D_r, or deletes last(D_f), based on whether last(D_f) < first(D_r), or not. While clean up is in progress, new arrivals are batched into a separate dirty queue DD. After the current clean up phase completes, the items of D_r are gradually transferred to C, until min(DD) ≤ first(D_r). At this point, the rest of the items of D_r are discarded, the contents of DD are transferred to D_f, and the next clean up phase is commenced. Thus, the total volume of

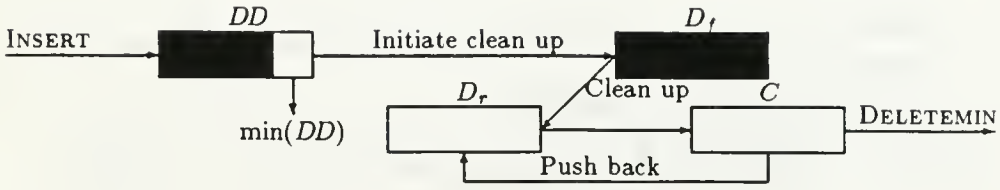


Figure 3: Worst case data structure 2.

residual clean up work left at any instant is at most $|DD| + |D_f|$. As before, the items of C are the smallest items among all items in the data structure, and $|C| \geq |DD| + |D_f|$. D_r also plays the role of the push back buffer. When the inserted item is smaller than some of the items of C, the queues DD,D_r, and D_f are all emptied, the items in rest(C) are pushed back into D_r, and the arriving item is put in DD. See figure 3.

Formally, the queues DD,D_r,D_f, and C satisfy the following conditions:

1. $C \parallel D_f \parallel D_r \parallel DD$ represents the PQA.
2. C and D_r are clean, and $C < D_f \parallel D_r \parallel DD$.
3. first(DD) = min(DD).
4. **Bias:** $|C| \geq |DD| + |D_f|$.

Operation BIAS takes the form:

```
BIAS ≡
    if D_f ≠ () then
      { Clean up step }
        if D_r = () cor first(D_r) > last(D_f) then
          PASSBACK(D_f,D_r)                          { decrease |D_f| }
        else DELETELAST(D_f)                         { decrease |D_f| }
    else if D_r ≠ () cand (DD = () cor first(D_r) < first(DD)) then
      PASS(D_r,C)                                    { increase |C| }
    else if DD ≠ () then
      { Destroy D_r; initiate a new clean up process }
        PASS(DD,C)                  { decrease |DD|; increase |C| }
        DD, D_r, D_f := (), (), DD
    { else D_f = D_r = DD = () }
```

BIAS increases $|C| - |DD| - |D_f|$ by at least 1, whenever at least one of DD,D_r, and D_f is nonempty. This is accomplished by either performing a clean up step, or passing an item from D_r to C, or initiating a new clean up phase. INSERT and

DELETEMIN require only one BIAS each, since both increase $|C| - |DD| - |D_f|$ by at most 1.

The operations of the PQA follow:

```
CREATEPQA ≡
    C, D_f, D_r, DD := (), (), (), ()

INSERT(x) ≡
    if C ≠ () cand first(C) ≥ x then
      { Destroy all queues; add x to C }
          C, D_f, D_r, DD := (x), (), (), ()
    else if C ≠ () cand last(C) ≥ x then
      { Destroy D_f, D_r, and DD; push back rest(C) into D_r;
        add x to DD }
          C, D_f, D_r, DD := (first(C)), (), rest(C), (x)
    else if DD ≠ () cand first(DD) ≥ x then
      { Destroy DD; add x to DD }
          DD := (x)
    else DD := DD || (x)
          BIAS

DELETEMIN ≡
    BIAS
    return DELETEFIRST(C)
        {BIAS ensures that C ≠ (), unless the PQA is empty }
```

This concludes the description of the second worst case data structure. It is easy to verify that the operations are performed correctly, and require only $O(1)$ time. DELETE can be added to the repertoire of operations as before.

5 Worst case data structure 3

Instead of cleaning up the dirty queue by scanning it in a fixed direction, we can perform the clean up steps at arbitrary positions in the middle of the dirty queue. For this, we partition the dirty queue D into a sequence of clean queues $D_1, D_2, \ldots, D_k$ such that, $D_1 \parallel D_2 \parallel \cdots \parallel D_k$ represents the PQA. A clean up step either combines two adjacent queues D_i and D_{i+1} into a single queue by concatenation, or deletes the last item of a queue D_i, or deletes an empty queue D_i, maintaining the queues clean. The amount of work needed to reduce all the queues into a single clean queue is at most $|D_1| + \cdots + |D_k| + k$, yielding the bias condition $|C| \geq |D_1| + \cdots + |D_k| + k$. A new item x is added by creating a singleton queue $Q_{k+1} = (x)$. A push back buffer B is needed as in worst case data structure 1. The details of the operations can be worked out as before.

6 Conclusion

The key to the success of our worst case data structures is the bias condition, which ensures that the query value (the minimum item, here) will be easily computable even though the construction of the data structure is not complete. The data structure invariants are chiefly responsible for managing the complex data movement elegantly.

Not every data structure with a good amortized performance has a counterpart with a matching worst case performance. The *Priority stack with attrition*(*PSA*) is a data structure whose only difference from the PQA is that INSERT now deletes the items that are smaller than or equal to the inserted item. Surprisingly, the PSA does not have an $O(1)$ worst case time data structure. To see this, consider the following neighbour search problem, where we are given n reals $x_1, x_2, \ldots, x_n$, and we want to preprocess them, and answer one query, which asks for the smallest real in the set larger than the query number x. Any data structure for this problem must spend $\Omega(\log(n))$ time on some query under the decision tree model. We can solve this problem by building a PSA of the numbers by inserting them in decreasing order (preprocessing) and then inserting x and performing two DELETEMINs (query processing). Thus, either INSERT or DELETEMIN costs logarithmic time, in any data structure for the PSA. This bound is tight, since the PSA has an implicit data structure, with $O(\log(d))$ time for INSERT, where d = the number of deleted items, and $O(1)$ time for DELETEMIN. Maintain the PSA contiguously in an array, such that the items are in key-order. INSERT performs an expanding binary search of the item list, starting from the smallest item, and inserts the new item at the right place. That is, the inserted item is successively compared to the smallest item, the second smallest item, the fourth smallest item, and so on, until a larger item is found. The exact position of the inserted item is located by a binary search of the sublist between the last two comparands.

Acknowledgement

I am grateful to Richard Cole and Bud Mishra for posing this problem and helpful discussions.

References

1. R.Cole and A.Siegel, River routing every which way, but loose, Proc. 25th IEEE FOCS, 1984, pp 65-73.

2. G.Diehr and B.Faaland, Optimal pagination of B-trees with variable length items, Comm. ACM 27, 1984, pp 241-247.

3. H.Gajewska and R.E.Tarjan, Deques with heap order, Information Processing Letters 22, 1986, pp 197-200.

4. L.L.Larmore and D.S.Hirschberg, Efficient optimal pagination of scrolls, Tech. Rept. #224, ICS Dept., Univ. of California at Irvine, CA, 1984.

5. E.M.McCreight, Pagination of B^*-trees with variable length records, Comm. ACM 20, 1977, pp 670-674.

NYU COMPSCI TR-385 c.2
Sundar, Rajamani
Worst case data structures
for the priority queue
with attrition.

NYU COMPSCI TR-385 c.2
Sundar, Rajamani
Worst case data structures
for the priority queue
with attrition.

This book may be kept JUL 15 1983
FOURTEEN DAYS
A fine will be charged for each day the book is kept overtime.

GAYLORD 142 PRINTED IN U S A